Role-based Interface Automata*

Extended abstract

Oana Andrei Muffy Calder Alice Miller
School of Computing Science, University of Glasgow, Glasgow, UK
firstname.lastname@glasgow.ac.uk

Component-based software system development is based on highly specialised components, each developed to meet different objectives, which are then composed into a system with an overall objective. We propose *Role-based interface automata* (RI automata) [2], a new formalism for modelling and reasoning about component-based systems. We focus on the actions exposed by *interfaces*, and their temporal ordering because, in general, we may not have access to individual component behaviour. This formalism extends Interface Automata [1] by defining *role-based actions* and component *objectives* in a temporal action logic.

A key feature of RI automata is the different types of transitions and the *roles* they represent. As with original interface automata, we distinguish between *input* for receiving, *output* for sending, and hidden (or internal) actions. Unlike interface automata, which conflates input with methods that can be called, and output with method calls, we further differentiate input and output actions according to three *roles*. These reflect fulfilment of component requirements and environment requirements, and are classified as: *master*, which calls (or requires), *servant*, which is callable (or supports, offers) and *valet*, which behaves like a servant or a silent (or anonymous hidden) action. Essentially a valet action is a choice that is determined by the environment. The valet does whatever is required by a master, if it can.

Action names are shared between component interfaces and the role each action has in a component depends upon that component. For example, when modelling a GPS-based navigation system, the transition label i.M?loc represents the action "component i requires the loc method (*get location*) and receives data (for example, GPS coordinates)". Another example is the transition label j.S!loc, which represents the action "component j offers the loc method, delivering data (e.g. GPS coordinates) to the requesting component." The former action has a master role (M), whereas the the latter has a servant role (S). When components are composed, denoted here by $\|$, they synchronise on shared actions so that input/output and role requirements are met. Informally, this means that in any composition, master actions must be synchronised with another servant or valet action (collectively known as subservient actions) in order to fulfil a component's objectives, whereas valet and servant actions are not subject to the same constraint. Non-synchronised subservient actions represent spare capacity. Role-based interface automata interact through four types of synchronisation: MI/VO, MI/SO, MO/VI, MO/SI, where M stands for master, V for valet, S for servant, I for input and O for output. For example the action i.M?loc can synchronise with j.S!loc, as an MI/SO synchronisation. Internal actions of the component automata are interleaved asynchronously, as usual for interface automata.

When composing two RI automata we use an optimistic or *busy* form of synchronisation. This means that if one automaton needs to perform a shared action but the other is not ready to reciprocate, we allow the first to wait until an appropriate action is offered; the latter automaton can only perform hidden actions in the meantime. If the second automaton can only follow paths in which the synchronising action is never offered, then the two automata are not compatible and the composition is not possible. RI

*This work was funded by the EPSRC grant, *Verifying Interoperability Requirements in Pervasive Systems* EP/F033206/1

automata are composed in two stages. Two RI automata are said to be *composable* if their actions are suitably disjoint, for example, output actions are disjoint, input actions are disjoint, etc.

We aim for a lightweight framework in which to express component objectives and check whether component replacements preserve the initial objectives: one that can be implemented on a small device (e.g. a mobile phone) and used in real-time, when required, for example, before a potential replacement component is downloaded. For instance, a phone application may evolve over time, replacing some components by cheaper ones, others by ones with more functionality, and yet others by simpler ones that work faster or have a more attractive user interface. We express component objectives as temporal properties, called *services*, in a fragment of ALTL [11, 8], which we call *Service-based Action Linear Temporal Logic* (SALTL), and we define an appropriate satisfaction relation.

Consider a software system composed of a component (or module) C and a context K (as composition of components) and let C' be a new component we want to replace C within context K. In the case of interface automata [1] it crucial that the new component (implementation) C' refines the behaviour of the old component (specification) C, i.e., C' has at least the same inputs (method definitions) and at most the same outputs (method calls) as C. For RI automata, the refinement relation between implementation and specification translates informally to: implementation must allow fewer master actions and more subservient actions than the specification. However, we propose that in component-based system development another approach, the *replacement* (or substitution) of one component by another, to be taken such that (i) the new component satisfies the same services as the old one and (ii) the new system still achieves its objectives. We consider replacement to be more liberal than refinement, i.e., it should not depend on the refinement relation between the old and the new component, but upon the old component and component and system objectives that are considered to be relevant. Let Q be the RI automaton modelling the interface of the context K above (which may be one or a composition of several RI automata), and P and P' the RI automata modelling the interfaces of the old component C and the new component C' respectively. We say that the RI automaton P' can replace the RI automaton P in the context Q with respect to some component objectives expressed as a service Σ_c and some system objectives expressed as a service Σ_s when:

- $(\mathbf{p_1})$ P' must satisfy all services Σ_c provided by P: *if* $P \models \Sigma_c$ *then* $P' \models \Sigma_c$;
- $(\mathbf{p_2})$ $P'\|Q$ must be a valid composition and provide all the services provided by $P\|Q$: P' *and* Q *are compatible and if* $P\|Q \models \Sigma_s$ *then* $P'\|Q \models \Sigma_s$.

For the problem $(\mathbf{p_1})$ we just have to solve the SALTL satisfaction problem $P' \models \Sigma_c$. The problem $(\mathbf{p_2})$ reduces to checking $P'\|Q \models \Sigma_s$, which may be a difficult problem to solve and it is not always appropriate to check it in real-time, in case $P'\|Q$ is a prohibitively large automaton. Therefore we adopt the approach of assume-guarantee reasoning frameworks [9, 12], and consider an *abstraction* of contexts as assumptions about the actions offered to or required from the component P. Then, we need only check that P' satisfies Σ_s under such assumptions. We note that an assume-guarantee reasoning framework has been defined for Interface Automata [7] based on the refinement relation between the old and the new component.

RI automata resemble modal I/O automata [10, 3] and modal specifications [13] due to the fact that they capture the *deontics* of actions. However, they are subtly different: RI automata emphasise the unique role that a component's action has in a composition. On the other hand, modal I/O automata and modal specifications label actions by modalities to indicate whether they should necessarily be included in an implementation, and play an important role in interface refinement. Our chief objective is that of replacement, rather than refinement, so this distinction is key.

Another formalism addressing the problems associated with component-based systems is Component-Interaction Automata [5], designed to allow for the specification of various aspects of hierarchical component-based systems and, specifically, to be used in combination with architecture description languages. Unlike our work, roles are not defined, so making the formalism less expressive.

An early version of RI automata, called PI automata (*Pervasive Interface* automata) is presented in [6]. A major difference between RI automata and PI automata is the latter only includes two types of action (master and slave), as well as employing a different notation. We made these developments as a consequence of our experiences of applying PI automata to a pervasive software system case study (the DOMINO framework [4] – a multiple-user mobile phone-based application). In [2], the definitions of RI automata composition (and composability and compatibility), and the service logic have been made more rigorous, and we have included examples illustrating the action roles and some compositions. Future work involves implementing a model checker for the satisfaction relation and a full assume-guarantee framework.

References

[1] Luca de Alfaro & Thomas A. Henzinger (2001): *Interface Automata*. In: ACM, editor: *ESEC / SIGSOFT FSE*. pp. 109–120.

[2] Oana Andrei, Muffy Calder & Alice Miller (2011): *Role-based Interface Automata*. Submitted to *Science of Computer Programming* `http://dcs.gla.ac.uk/~oandrei/docs/RIA-submitted-SCP11.pdf`.

[3] Sebastian S. Bauer, Philip Mayer, Andreas Schroeder & Rolf Hennicker (2010): *On Weak Modal Compatibility, Refinement, and the MIO Workbench*. In: Javier Esparza & Rupak Majumdar, editors: *TACAS*, *LNCS* 6015. Springer, pp. 175–189.

[4] Marek Bell, Malcolm Hall, Matthew Chalmers, Philip D. Gray & Barry Brown (2006): *Domino: Exploring Mobile Collaborative Software Adaptation*. In: Kenneth P. Fishkin, Bernt Schiele, Paddy Nixon & Aaron J. Quigley, editors: *Proceedings of PERVASIVE 2006*, *LNCS* 3968. Springer, pp. 153–168.

[5] Lubos Brim, Ivana Cerná, Pavlína Vareková & Barbora Zimmerova (2006): *Component-interaction automata as a verification-oriented component-based system specification*. *ACM SIGSOFT Software Engineering Notes* 31(2).

[6] Muffy Calder, Phil Gray, Alice Miller & Chris Unsworth (2011): *An Introduction to Pervasive Automata*. In: *Proceedings of FACS 2010*, *LNCS* 6921. Springer, pp. 71–87.

[7] Michael Emmi, Dimitra Giannakopoulou & Corina S. Pasareanu (2008): *Assume-Guarantee Verification for Interface Automata*. In: Jorge Cuéllar, T. S. E. Maibaum & Kaisa Sere, editors: *Proceedings of FM 2008*, *LNCS* 5014. Springer, pp. 116–131.

[8] Dimitra Giannakopoulou & Jeff Magee (2003): *Fluent model checking for event-based systems*. In: *ESEC / SIGSOFT FSE*. ACM, pp. 257–266.

[9] Cliff B. Jones (1983): *Specification and Design of (Parallel) Programs*. In: *IFIP Congress*, 321–332.

[10] Kim G. Larsen, Ulrik Nyman & Andrzej Wasowski (2007): *Modal I/O Automata for Interface and Product Line Theories*. In: Rocco De Nicola, editor: *ESOP*, *LNCS* 4421. Springer, pp. 64–79.

[11] R. De Nicola, A. Fantechi, S. Gnesi & G. Ristori (1993): *An action-based framework for veryfying logical and behavioural properties of concurrent systems*. *Computer Networks and ISDN Systems* 25(7), pp. 761–778.

[12] Amir Pnueli (1985): *In transition from global to modular temporal reasoning about programs*. In: *Logics and models of concurrent systems*. Springer-Verlag New York, Inc., pp. 123–144.

[13] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay & Roberto Passerone (2011): *A Modal Interface Theory for Component-based Design*. *Fundam. Inform.* 108(1–2), pp. 119–149.